CGA Call Graph Analyzer – Locating and Understanding Functionality within the Gnu Compiler Collection's Million Lines of Code

Johannes Bohnet, Jürgen Döllner
Hasso-Plattner-Institut – University of Potsdam
{bohnet, doellner}@hpi.uni-potsdam.de

Abstract

In this paper we describe the application of our tool (CGA) for locating and understanding functionality in unfamiliar code of complex software systems onto the Gnu Compiler Collection GCC (approx. 1 million lines of C-code). The analysis' goal is to identify and understand those code locations that implement GCC's functionality of 'parsing constructors in C++ programs'.

The Analysis Process with CGA

We apply our *CGA* analysis tool on the *GCC v4.1.0*. In a first step, *GCC*'s usual Linux build process with an additional compiler flag '-g' is performed to build all its executables and libraries with debug information. In a second step, *GCC*'s module structure is reconstructed by parsing the directory structure of the code repository with the *CGA* tool. The automatically reconstructed structure is thereupon manually checked for directory clones. For the *GCC*, however, the proposed module structure seems to be reasonable and does not need to be restructured.

Next, the GCC is executed in a way that the functionality of 'parsing constructors in C++ programs' is executed. For this, a small test program is compiled with *GCC*'s *g++ compiler*. To obtain the graph of called functions during execution, we run *g++* within the profiling suite *Valgrind/Callgrind*:

```
callgrind g++ -c foo.cpp
```

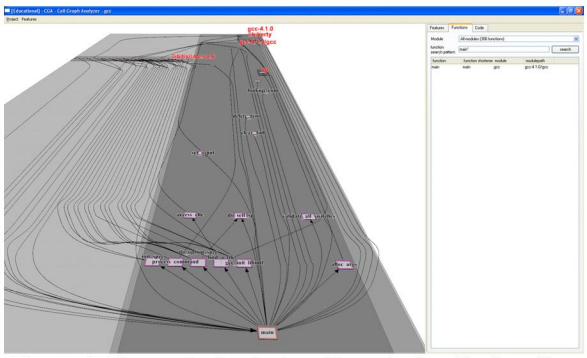

Figure 1: The analysis starts with exploring the function calls induced by *g++*'s main function.

The test program *foo.cpp* implements a simple class:

```
class Foo {
     int val;
     Foo() {val = 10;}
};
```

Finally, the function call graph is visualized within our *CGA* tool. With the search feature of our tool, *g++*'s main function is identified and used as entry point for call graph analysis (Figure 1).

The *CGA* tool shows a subgraph of the complete function call graph consisting approx. 50 functions that are connected to *main()* via calls. Most of them lie within the module *gcc-4.1.0/gcc*. The function *process_command()* seems to be a good candidate that leads to *GCC*'s 'C++ parsing' functionality. Therefore, we double-click on this function and obtain (a) a graph visualization showing functions that are closely connected to *process_command()* via calls and (b) the function's source code (Figure 2). Within the source code view, each executed code line is highlighted. Additionally, calls that are hidden by macros or function pointers are resolved to concrete calls. This

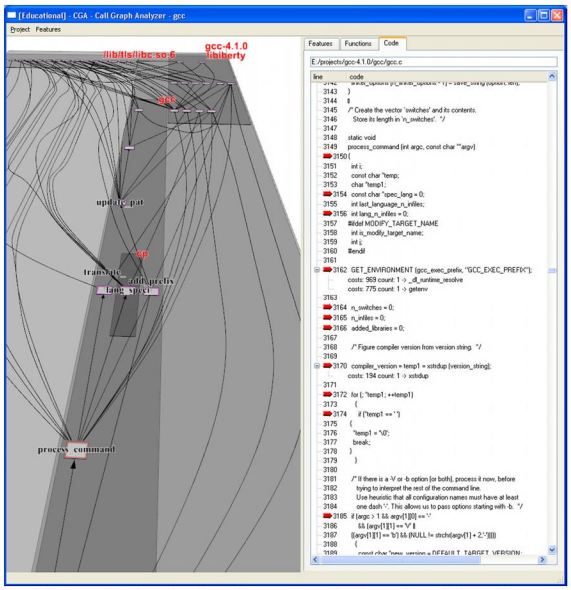

Figure 2: Analyzing the executed code of function *process_command()* reveals that this function is not relevant for finding *GCC*'s c++ parsing functionality.

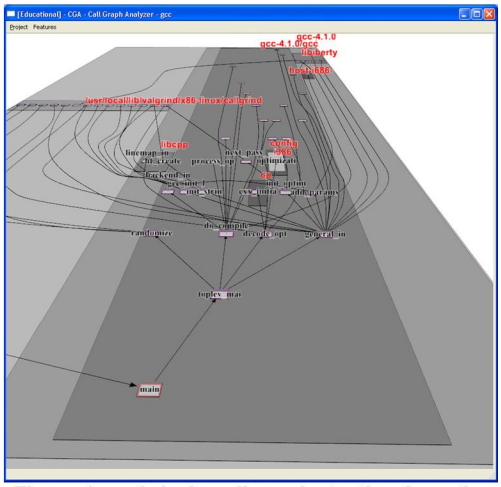

Figure 3: *cc1plus*'s call graph starting from the main function.

allows efficient reading of relevant code parts. Analyzing the code yields the result that *process_command()* is not relevant for finding *GCC*'s 'C++ parsing' functionality.

Next, we step back to *main()* and then analyze *do_option_spec()*, which leads to the function *execute()*, which sounds interesting. Hence, *execute()* is double-clicked and analyzed in detail. From following function calls that lead to the *gcc4.1.0/libiberty* module and from reading code, we understand that the *GCC* (a) starts new processes for each language specific compiler and (b) uses pipes for inter-process communication. Therefore, we run *g++* again and trace the execution with the Linux command *strace*. With it, we discover that a child process is

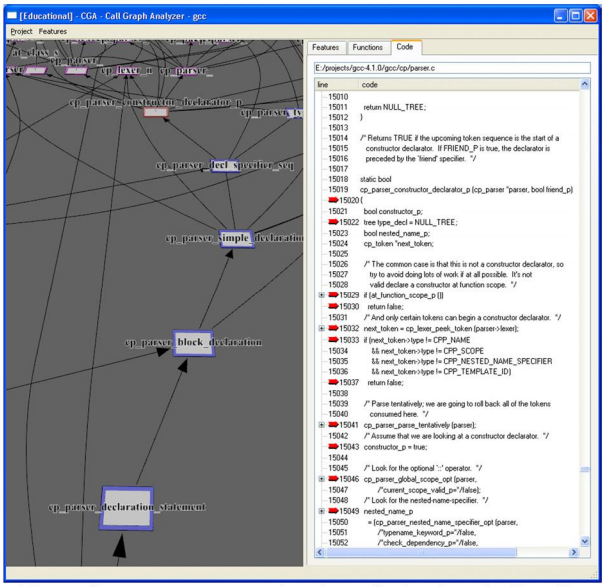

Figure 4: *cp_parser_constructor_p()* checks whether the currently parsed token sequence is a C++ constructor.

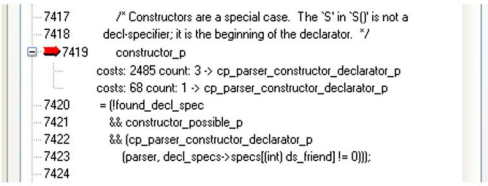

Figure 5: *cp_parser_constructor_p()* is called 4 times. This corresponds to the number of substructures in the test program *foo.cpp*.

created executing the *cc1plus* binary file.

Next, the function call graph of this child process is logged with *Callgrind/Valgrind*. Figure 3 shows the call graph starting from *cc1plus*'s main function. By reading the function names and double-clicking on successive functions, we rapidly find the function *c_parse_file()* located in the module *gcc4.1.0/gcc/cp*:

main()	module: *gcc4.1.0/gcc*
→ *toplev_main()*	module: *gcc4.1.0/gcc*
→ *do_compile()*	module: *gcc4.1.0/gcc*
→ *compile_file()*	module: *gcc4.1.0/gcc*
→ *c_common_parse_file()*	module: *gcc4.1.0/gcc*
→ *c_parse_file()*	module: *gcc4.1.0/gcc/cp*

Within the *gcc4.1.0/gcc/cp* module we identify by following calls and by reading code that function *cp_parser_constructor_declarator_p()* implements parsing a C++ constructor. We additionally understand the chain of calls to this function (Figure 4):

c_parse_file() → *cp_parser_translation_unit()*
→ *cp_parser_declaration_seq_opt()* → *cp_parser_declaration()*
→ *cp_parser_block_declaration()* → *cp_parser_simple_declaration()*
→ *cp_parser_decl_specifier_seq()* → *cp_parser_constructor_declaration_p()*.

An additional finding is that this function is called 4 times; 3 times not having found a constructor and once having found a constructor (Figure 5). This corresponds to the code of the test program *foo.cpp*.

Conclusions

The *CGA* tool supported us in rapidly finding *GCC*'s functionality of 'parsing C++ constructors' within a large amount of unfamiliar code. *CGA* first helped us to understand *GCC*'s inter-process communication with pipes for performing language specific compilations. This led to an analysis of the dynamics of the *cc1plus* executable. With *CGA* we identified the function that implements parsing C++ constructors and we were able to understand the context, i.e., the chains of successive calls, in which this function is used. The analysis took about 2 hours without having had any knowledge on the *GCC* implementation before. The most important features of *CGA* for the analysis of *GCC* were (a) the overview-like visualization of function interaction within the module structure and (b) the ease of switching to code when information gathered from call graphs was too imprecise for building up an understanding of the system implementation.